410250: High Performance Computing
Group A
Assignment No.: 1

Title of the Assignment: Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS

Objective of the Assignment: Students should be able to Write a program to implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP

Prerequisite:

1. Basic of programming language
2. Concept of BFS and DFS
3. Concept of Parallelism

Contents for Theory:

1. What is BFS?
2. What is DFS?
3. Concept of OpenMP
4. Code Explanation with Output

What is BFS?

BFS stands for Breadth-First Search. It is a graph traversal algorithm used to explore all the nodes of a graph or tree systematically, starting from the root node or a specified starting point, and visiting all the neighbouring nodes at the current depth level before moving on to the next depth level. The algorithm uses a queue data structure to keep track of the nodes that need to be visited, and marks each visited node to avoid processing it again. The basic idea of the BFS algorithm is to visit all the nodes at a given level before moving on to the next level, which ensures that all the nodes are visited in breadth-first order. BFS is commonly used in many applications, such as finding the shortest path between two nodes, solving puzzles, and searching through a tree or graph.

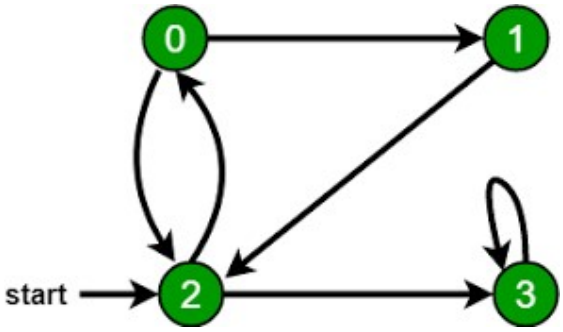
Now let's take a look at the steps involved in traversing a graph by using Breadth-First Search:

Step 1: Take an Empty Queue.

Step 2: Select a starting node (visiting a node) and insert it into the Queue.

Step 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

Step 4: Print the extracted node.



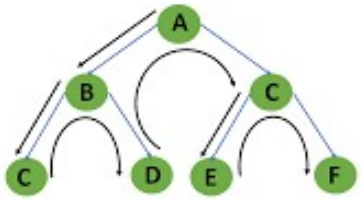
What is DFS?

DFS stands for Depth-First Search. It is a popular graph traversal algorithm that explores as far as possible along each branch before backtracking. This algorithm can be used to find the shortest path between two vertices or to traverse a graph in a systematic way. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. The backtracking is done to explore the next branch that has not been explored yet.

DFS can be implemented using either a recursive or an iterative approach. The recursive approach is simpler to implement but can lead to a stack overflow error for very large graphs. The iterative approach uses a stack to keep track of nodes to be explored and is preferred for larger graphs.

DFS can also be used to detect cycles in a graph. If a cycle exists in a graph, the DFS algorithm will eventually reach a node that has already been visited, indicating that a cycle exists. A standard DFS implementation puts each vertex of the graph into one of two categories: 1. Visited 2. Not Visited The

purpose of the algorithm is to mark each vertex as visited while avoiding cycles.



Concept of OpenMP

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

● OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

● OpenMP is widely used in scientific computing, engineering, and other fields that require high-performance computing. It is supported by most modern compilers and is available on a wide range of platforms, including desktops, servers, and supercomputers. How Parallel BFS Work .

● Parallel BFS (Breadth-First Search) is an algorithm used to explore all the nodes of a graph or tree SNJB's Late Sau.K.B.Jain College of Engineering Chandwad 3 Department of Computer Engineering Course: Laboratory Practice V systematically in parallel. It is a popular parallel algorithm used for graph traversal in distributed computing, shared-memory systems, and parallel clusters.

● The parallel BFS algorithm starts by selecting a root node or a specified starting point, and then assigning it to a thread or processor in the system. Each thread maintains a local queue of nodes to be visited and marks each visited node to avoid processing it again.

● The algorithm then proceeds in levels, where each level represents a set of nodes that are at a certain distance from the root node. Each thread processes the nodes in its local queue at the current level, and then exchanges the nodes that are adjacent to the current level with other threads or processors. This is done to ensure that the nodes at the next level are visited by the next iteration of

the algorithm.

● The parallel BFS algorithm uses two phases: the computation phase and the communication phase. In the computation phase, each thread processes the nodes in its local queue, while in the communication phase, the threads exchange the nodes that are adjacent to the current level with other threads or processors.

● The parallel BFS algorithm terminates when all nodes have been visited or when a specified node has been found. The result of the algorithm is the set of visited nodes or the shortest path from the root node to the target node.

● Parallel BFS can be implemented using different parallel programming models, such as OpenMP, MPI, CUDA, and others. The performance of the algorithm depends on the number of threads or processors used, the size of the graph, and the communication overhead between the threads or processors

Conclusion:

In this way we can achieve parallelism while implementing Breadth First Search and Depth First Search.

Assignment No.: 2

Title of the Assignment: Write a program to implement Parallel Bubble Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

Objective of the Assignment: Students should be able to Write a program to implement Parallel Bubble Sort and can measure the performance of sequential and parallel algorithms.

Prerequisite:

1. Basic of programming language
2. Concept of Bubble Sort
3. Concept of Parallelism

Contents for Theory:

1. What is Bubble Sort? Use of Bubble Sort
2. Example of Bubble sort?
3. Concept of OpenMP
4. How Parallel Bubble Sort Work
5. How to measure the performance of sequential and parallel algorithms?

What is Bubble Sort?

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is called "bubble" sort because the algorithm moves the larger elements towards the end of the array in a manner that resembles the rising of bubbles in a liquid.

The basic algorithm of Bubble Sort is as follows:

1. Start at the beginning of the array.
2. Compare the first two elements. If the first element is greater than the second element, swap them.
3. Move to the next pair of elements and repeat step 2.
4. Continue the process until the end of the array is reached.

5. If any swaps were made in step 2-4, repeat the process from step 1.

The time complexity of Bubble Sort is O(n^2), which makes it inefficient for large lists. However, it has the advantage of being easy to understand and implement, and it is useful for educational purposes and for sorting small datasets.

Bubble Sort has limited practical use in modern software development due to its inefficient time complexity of O(n^2) which makes it unsuitable for sorting large datasets. However, Bubble Sort has some advantages and use cases that make it a valuable algorithm to understand, such as:

1. Simplicity: Bubble Sort is one of the simplest sorting algorithms, and it is easy to understand and implement. It can be used to introduce the concept of sorting to beginners and as a basis for more complex sorting algorithms.

2. Educational purposes: Bubble Sort is often used in academic settings to teach the principles of sorting algorithms and to help students understand how algorithms work.

3. Small datasets: For very small datasets, Bubble Sort can be an efficient sorting algorithm, as its overhead is relatively low.

4. Partially sorted datasets: If a dataset is already partially sorted, Bubble Sort can be very efficient. Since Bubble Sort only swaps adjacent elements that are in the wrong order, it has a low number of operations for a partially sorted dataset.

5. Performance optimization: Although Bubble Sort itself is not suitable for sorting large datasets, some of its techniques can be used in combination with other sorting algorithms to optimize their performance. For example, Bubble Sort can be used to optimize the performance of Insertion Sort by reducing the number of comparisons needed.

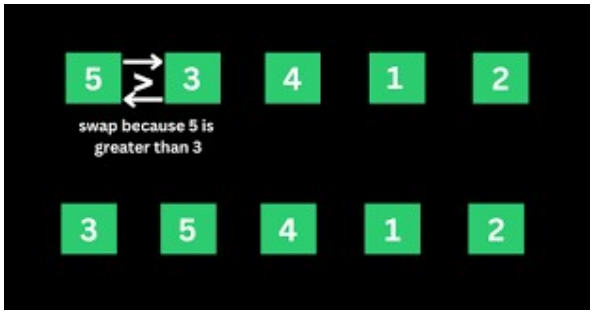
Example of Bubble sort:

Let's say we want to sort a series of numbers 5, 3, 4, 1, and 2 so that they are arranged in ascending order…

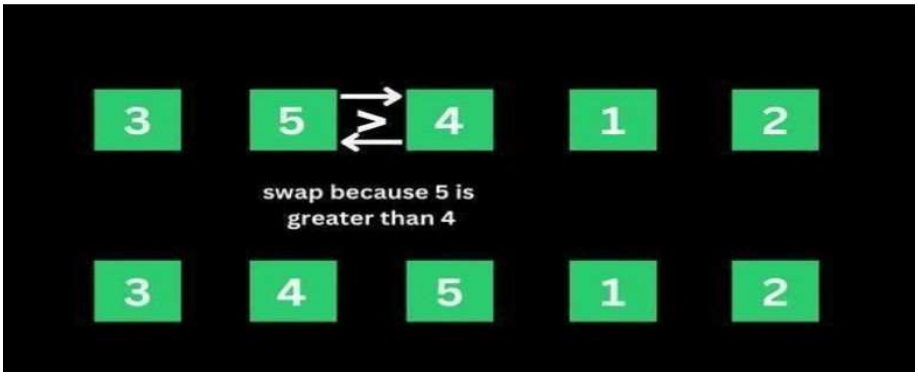
The sorting begins the first iteration by comparing the first two values. If the first value is greater than the second, the algorithm pushes the first value to the index of the second value.

First Iteration of the Sorting

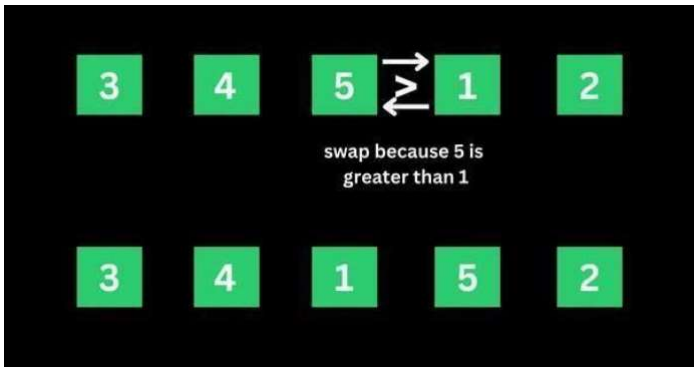
Step 1: In the case of 5, 3, 4, 1, and 2, 5 is greater than 3. So 5 takes the position of 3 and the numbers become 3, 5, 4, 1, and 2.



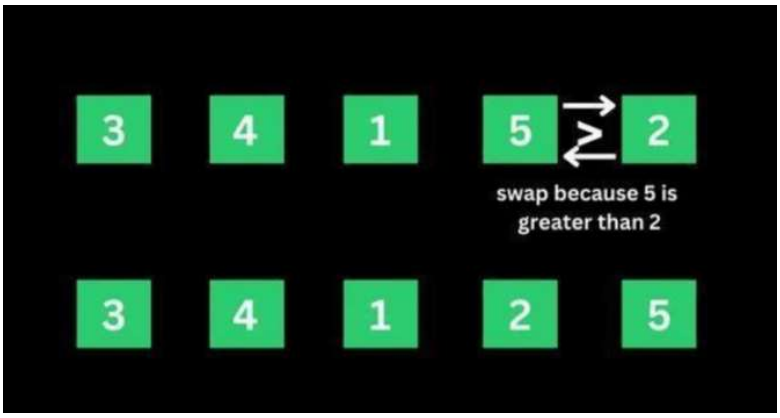
Step 2: The algorithm now has 3, 5, 4, 1, and 2 to compare, this time around, it compares the next two values, which are 5 and 4. 5 is greater than 4, so 5 takes the index of 4 and the values now become 3, 4, 5



Step 3: The algorithm now has 3, 4, 5, 1, and 2 to compare. It compares the next two values, which are 5 and 1. 5 is greater than 1, so 5 takes the index of 1 and the numbers become 3, 4, 1, 5, and 2.



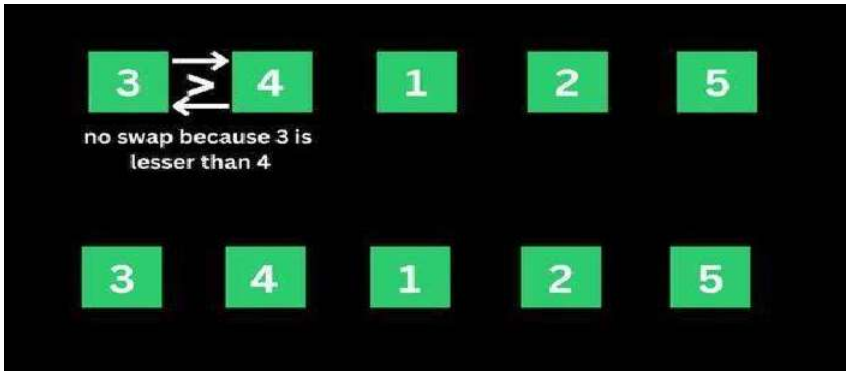
Step 4: The algorithm now has 3, 4, 1, 5, and 2 to compare. It compares the next two values, which are 5 and 2. 5 is greater than 2, so 5 takes the index of 2 and the numbers become 3, 4, 1, 2, and 5.



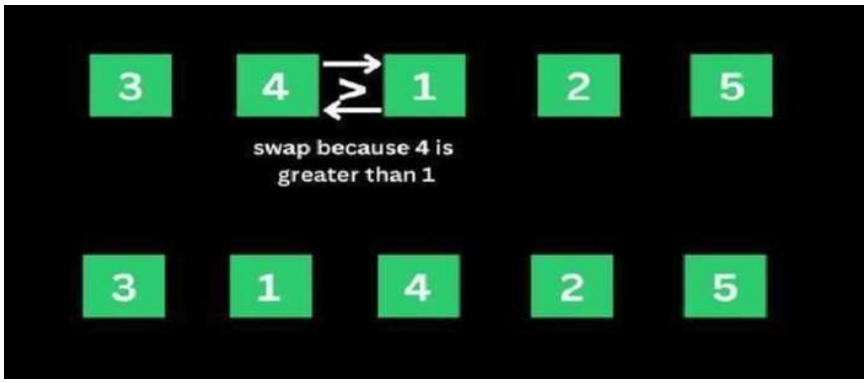
That's the first iteration. And the numbers are now arranged as 3, 4, 1, 2, and 5 – from the initial 5, 3, 4, 1, and 2. As you might realize, 5 should be the last number if the numbers are sorted in ascending order. This means the first iteration is really completed.

Second Iteration of the Sorting and the Rest

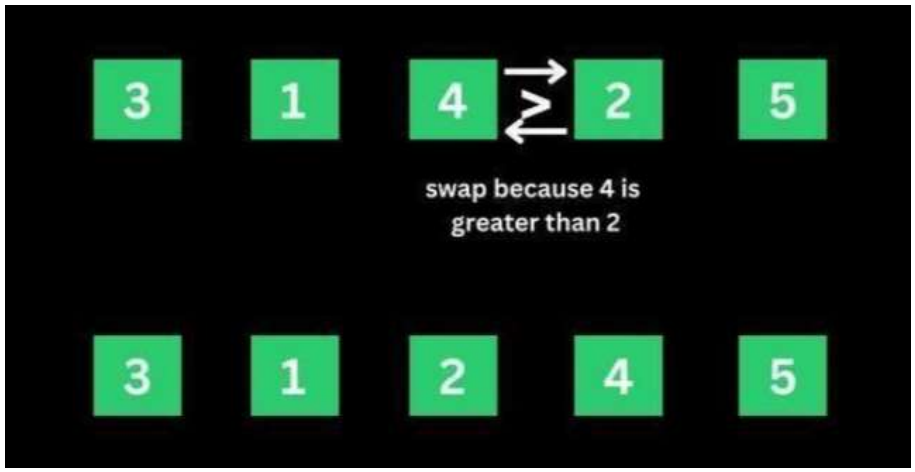
The algorithm starts the second iteration with the last result of 3, 4, 1, 2, and 5. This time around, 3 is smaller than 4, so no swapping happens. This means the numbers will remain the same.



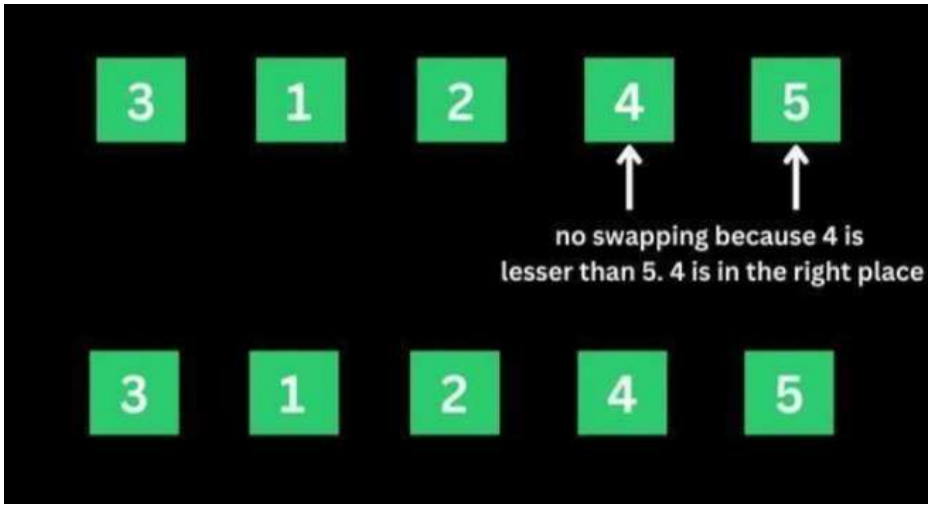
The algorithm proceeds to compare 4 and 1. 4 is greater than 1, so 4 is swapped for 1 and the numbers become 3, 1, 4, 2, and 5.



The algorithm now proceeds to compare 4 and 2. 4 is greater than 2, so 4 is swapped for 2 and the numbers become 3, 1, 2, 4, and 5.



4 is now in the right place, so no swapping occurs between 4 and 5 because 4 is smaller than 5.



That's how the algorithm continues to compare the numbers until they are arranged in ascending order of 1, 2, 3, 4, and 5.

Concept of OpenMP:

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution.

These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs.

● The compiler then generates parallel code that can run on multiple processors concurrently.

● OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

How Parallel Bubble Sort Work

● Parallel Bubble Sort is a modification of the classic Bubble Sort algorithm that takes advantage of parallel processing to speed up the sorting process.

● In parallel Bubble Sort, the list of elements is divided into multiple sub lists that are sorted concurrently by multiple threads. Each thread sorts its sub list using the regular Bubble Sort algorithm. When all sub lists have been sorted, they are merged together to form the final sorted list.

● The parallelization of the algorithm is achieved using OpenMP, a programming API that supports parallel processing in C++, Fortran, and other programming languages. OpenMP provides a set of compiler directives that allow developers to specify which parts of the code can be executed in parallel.

● In the parallel Bubble Sort algorithm, the main loop that iterates over the list of elements is divided into multiple iterations that are executed concurrently by multiple threads. Each thread sorts a subset of the list, and the threads synchronize their work at the end of each iteration to ensure that the elements are properly ordered.

● Parallel Bubble Sort can provide a significant speedup over the regular Bubble Sort algorithm, especially when sorting large datasets on multi-core processors. However, the speedup is limited by the overhead of thread creation and synchronization, and it may not be worth the effort for small datasets or when using a single-core processor.

How to measure the performance of sequential and parallel algorithms?

To measure the performance of sequential Bubble sort and parallel Bubble sort algorithms, you can follow these steps:

1. Implement both the sequential and parallel Bubble sort algorithms.
2. Choose a range of test cases, such as arrays of different sizes and different degrees of sortedness, to test the performance of both algorithms.
3. Use a reliable timer to measure the execution time of each algorithm on each test case.
4. Record the execution times and analyse the results.

When measuring the performance of the parallel Bubble sort algorithm, you will need to specify the number of threads to use. You can experiment with different numbers of threads to find the optimal value for your system.

How to check CPU utilization and memory consumption in ubuntu

In Ubuntu, you can use a variety of tools to check CPU utilization and memory consumption. Here are some common tools:

1. top: The top command provides a real-time view of system resource usage, including CPU utilization and memory consumption. To use it, open a terminal window and type top. The output will display a list of processes sorted by resource usage, with the most resource-intensive processes at the top.

2. htop: htop is a more advanced version of top that provides additional features, such as interactive process filtering and a color-coded display. To use it, open a terminal window and type htop.

3. ps: The ps command provides a snapshot of system resource usage at a particular moment in time. To use it, open a terminal window and type ps aux. This will display a list of all running processes and their resource usage.

4. free: The free command provides information about system memory usage, including total, used, and free memory. To use it, open a terminal window and type free -h.

5. vmstat: The vmstat command provides a variety of system statistics, including CPU utilization, memory usage, and disk activity. To use it, open a terminal window and type vmstat.

Conclusion:

In this way we can implement Bubble Sort in parallel way using OpenMP also come to know how to how to measure performance of serial and parallel algorithm.

Assignment No.: 3

Title of the Assignment: Implement Min, Max, Sum and Average operations using Parallel Reduction.

Objective of the Assignment: Students should be able to learn about how to perform min, max, sum, and average operations on a large set of data using parallel reduction technique in CUDA. The program defines four kernel functions, reduce_min, reduce_max, reduce_sum, and reduce_avg.

Prerequisite:

1. Knowledge of parallel programming concepts and techniques, such as shared memory, threads, and synchronization.
2. Familiarity with a parallel programming library or framework, such as OpenMP, MPI, or CUDA.
3. A suitable parallel programming environment, such as a multi-core CPU, a cluster of computers, or a GPU.
4. A programming language that supports parallel programming constructs, such as C, C++, Fortran, or Python.

Contents of Theory:

Parallel Reduction Operation:

Parallel reduction is a common technique used in parallel computing to perform a reduction operation on a large dataset. A reduction operation combines a set of values into a single value, such as computing the sum, maximum, minimum, or average of the values. Parallel reduction exploits the parallelism available in modern multicore processors, clusters of computers, or GPUs to speed up the computation.

The parallel reduction algorithm works by dividing the input data into smaller chunks that can be processed independently in parallel. Each thread or process computes the reduction operation on its local chunk of data, producing a partial result. The partial results are then combined in a hierarchical manner until a single result is obtained.

The most common parallel reduction algorithm is the binary tree reduction algorithm, which has a logarithmic time complexity and can achieve optimal parallel efficiency. In this algorithm, the input data is initially divided into chunks of size n, where n is the number of parallel threads or processes. Each thread or process computes the reduction operation on its chunk of data, producing n partial results.

The partial results are then recursively combined in a binary tree structure, where each internal node represents the reduction operation of its two child nodes. The tree structure is built in a bottom-up manner, starting from the leaf nodes and ending at the root node. Each level of the tree reduces the number of partial results by a factor of two, until a single result is obtained at the root node.

The binary tree reduction algorithm can be implemented using various parallel programming models, such as OpenMP, MPI, or CUDA. In OpenMP, the algorithm can be implemented using the parallel and for directives for parallelizing the computation, and the reduction clause for combining the partial results. In MPI, the algorithm can be implemented using the MPI_Reduce function for performing the reduction operation, and the MPI_Allreduce function for distributing the result to all processes. In CUDA, the algorithm can be implemented using the parallel reduction kernel, which uses shared memory to store the partial results and reduce the memory access latency.

Parallel reduction has many applications in scientific computing, machine learning, data analytics, and computer graphics. It can be used to compute the sum, maximum, minimum, or average of large datasets, to perform data filtering, feature extraction, or image processing, to solve optimization problems, or to accelerate numerical simulations. Parallel reduction can also be combined with other parallel algorithms, such as parallel sorting, searching, or matrix operations, to achieve higher performance and scalability.

Conclusion:

In each section, we use a loop and a critical section to combine the maximum or sum values from each thread. The #pragma omp flush directive ensures that the values are properly synchronized between threads.

Assignment No.: 4

Title of the Assignment: Write a CUDA Program for:

1. Addition of two large vectors
2. Matrix Multiplication using CUDA

Objective of the Assignment: Students should be able to learn about parallel computing and students should learn about CUDA (Compute Unified Device Architecture) and how it helps to boost high performance computations.

Prerequisite:

1. Basics of CUDA Architecture.
2. Basics of CUDA programming model.
3. CUDA kernel function.
4. CUDA thread organization

Contents of Theory:

1. CUDA architecture: CUDA is a parallel computing platform and programming model developed by NVIDIA. It allows developers to use the power of GPU (Graphics Processing Unit) to accelerate computations. CUDA architecture consists of host and device components, where the host is the CPU and the device is the GPU.

2. CUDA programming model: CUDA programming model consists of host and device codes. The host code runs on the CPU and is responsible for managing the GPU memory and launching the kernel functions on the device. The device code runs on the GPU and performs the computations.

3. CUDA kernel function: A CUDA kernel function is a function that is executed on the GPU. It is defined with the global keyword and is called from the host code using a launch configuration. Each kernel function runs in parallel on multiple threads, where each thread

performs the same operation on different data.

4. Memory management in CUDA: In CUDA, there are three types of memory: global, shared, and local. Global memory is allocated on the device and can be accessed by all threads. Shared memory is allocated on the device and can be accessed by threads within a block. Local memory is allocated on each thread and is used for temporary storage.

5. CUDA thread organization: In CUDA, threads are organized into blocks, and blocks are organized into a grid. Each thread is identified by a unique thread index, and each block is identified by a unique block index.

6. Matrix multiplication: Matrix multiplication is a fundamental operation in linear algebra. It involves multiplying two matrices and producing a third matrix. The resulting matrix has dimensions equal to the number of rows of the first matrix and the number of columns of the second matrix.

CUDA stands for Compute Unified Device Architecture. It is a parallel computing platform and programming model developed by NVIDIA.CUDA allows developers to use the power of the GPU to accelerate computations. It is designed to be used with C, C++, and Fortran programming languages' architecture consists of host and device components. The host is the CPU, and the device is the GPU. The CPU is responsible for managing the GPU memory and launching the kernel functions on the device.

A CUDA kernel function is a function that is executed on the GPU. It is defined with the global keyword and is called from the host code using a launch configuration. Each kernel function runs in parallel on multiple threads, where each thread performs the same operation on different data.

CUDA provides three types of memory: global, shared, and local. Global memory is allocated on the device and can be accessed by all threads. Shared memory is allocated on the device and can be accessed by threads within a block. Local memory is allocated on each thread and is used for temporary storage.

CUDA threads are organized into blocks, and blocks are organized into a grid. Each thread is

identified by a unique thread index, and each block is identified by a unique block index.

CUDA devices have a hierarchical memory architecture consisting of multiple memory levels, including registers, shared memory, L1 cache, L2 cache, and global memory.

CUDA supports various libraries, including cuBLAS for linear algebra, cuFFT for Fast Fourier Transform, and cuDNN for deep learning.

CUDA programming requires a compatible NVIDIA GPU and an installation of the CUDA Toolkit, which includes the CUDA compiler, libraries, and tools.

CUDA Program for Matrix Multiplication:

This program multiplies two matrices of size n using CUDA. It first allocates host memory for the matrices and initializes them. Then it allocates device memory and copies the matrices to the device. It sets the kernel launch configuration and launches the kernel function matrix_multiply. The kernel function performs the matrix multiplication and stores the result in matrix c. Finally, it copies the result back to the host and frees the device and host memory.

The kernel function calculates the row and column indices of the output matrix using the block index and thread index. It then uses a for loop to calculate the sum of the products of the corresponding elements in the input matrices. The result is stored in the output matrix.

Note that in this program, we use CUDA events to measure the elapsed time of the kernel function. This is because the kernel function runs asynchronously on the GPU, so we need to use events to synchronize the host and device and measure the time accurately.

Conclusion:

Hence, we have implemented Addition of two large vectors and Matrix Multiplication using CUDA.